Parallel Algorithms

Prof. Michele Amoretti

High Performance Computing 2022/2023

Outline

- Modeling approaches
- Performance
- Design patterns

Modeling approaches

RAM model of computation

A convenient approach to design a **sequential algorithm** is to formulate it using an abstract model of computation called the **random-access machine (RAM)** model.

In this model, **the machine consists of a single processor connected to a memory system**.

Each basic CPU operation, including arithmetic operations, logical operations, and memory accesses, **requires one time step**.

The designer's goal is to develop an algorithm with modest time and memory requirements. The random-access machine model allows the algorithm designer to ignore many of the details of the computer on which the algorithm will ultimately be executed, but captures enough detail that the designer can predict with reasonable accuracy how the algorithm will perform.

Modeling parallel computations

Modeling **parallel computations** is more complicated than modeling sequential computations because parallel computers are more complex and variegated in organization than sequential computers.

As a consequence, a large portion of the research on parallel algorithms has gone into the question of modeling, and many debates have raged over what the “right” model is, or about how practical various models are.

For parallel computations, there are two major classes of models:

- **multiprocessor models**
- **work-depth models**

Multiprocessor models

A multiprocessor model is a generalization of the sequential RAM model in which there is more than one processor.

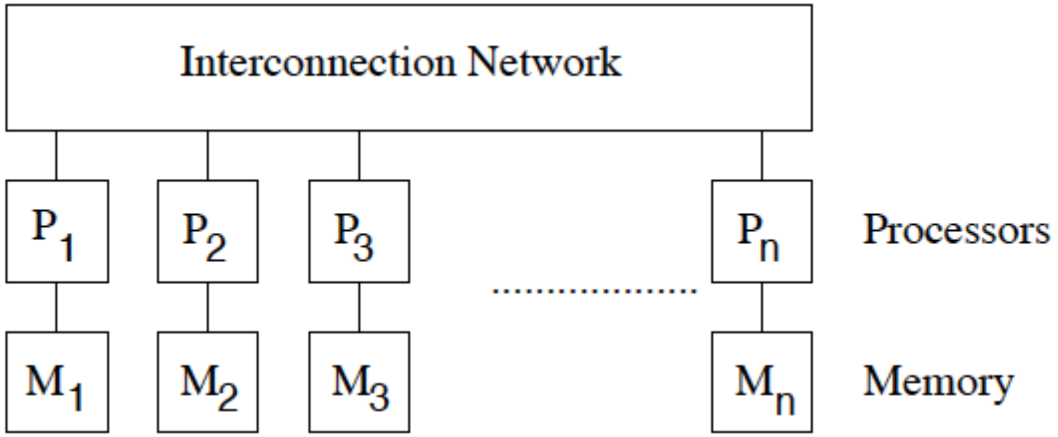
Three basic types:

- **local memory machine model**
- **modular memory machine model**
- **parallel random-access machine (PRAM) model**

Local memory machine model

In a local memory machine model, each processor can access its own local memory directly, but can access the memory in another processor only by sending a memory request through the network.

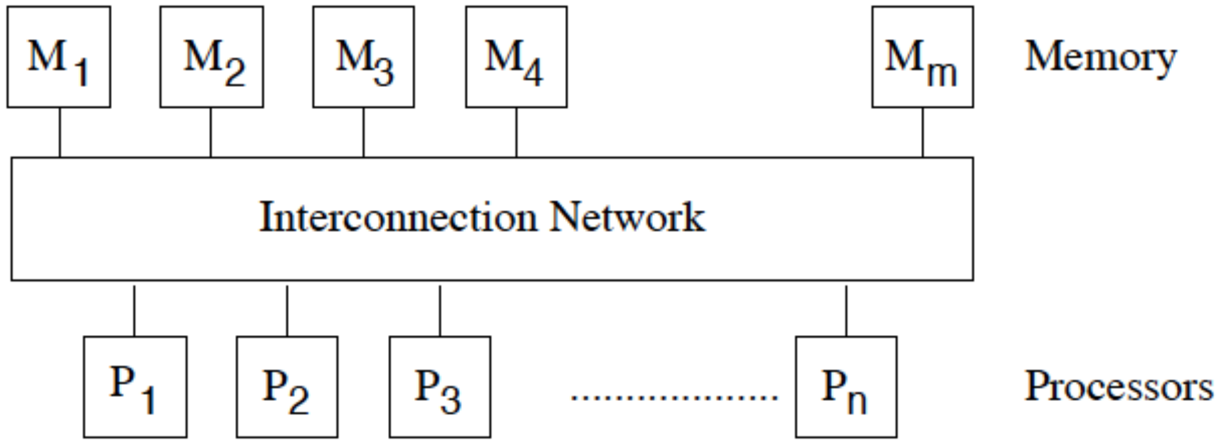
As in the RAM model, all local operations, including local memory accesses, take unit time. The time taken to access the memory in another processor, however, will depend on both the capabilities of the communication network and the pattern of memory accesses made by other processors, since these other accesses could congest the network.



Modular memory machine model

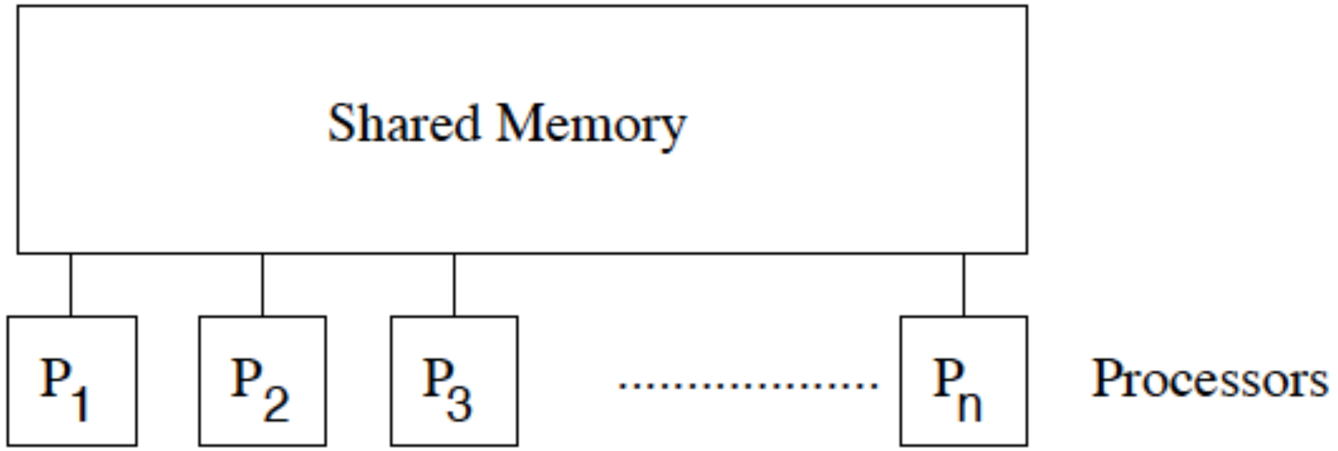
In a modular memory machine model, a processor accesses the memory in a memory module by sending a memory request through the network.

Typically the processors and memory modules are arranged so that the time for any processor to access any memory module is roughly uniform. As in a local memory machine model, the exact amount of time depends on the communication network and the memory access pattern.



PRAM model

In a PRAM model, a processor can access any word of memory in a single step. Furthermore, these accesses can occur in parallel, i.e., in a single step, every processor can access the shared memory.



PRAM model

The **synchronous** PRAM model has a similarity with data-parallel execution on a SIMD machine
• All processors execute the same program
• All processors execute the same PRAM step instruction stream in "lockstep"
• The effect of an operation depends on local data
• Instructions can be selectively disabled (for if-then-else flow)

The **asynchronous** PRAM model
• Several competing programs
• No lockstep

PRAM model

Exclusive Read (ER) - at any given step, only 1 processor can read a location during the same step.

Concurrent Read (CR) - no restriction on reading.

Exclusive Write (EW) - at any given step, only 1 processor can write a location during the same step.

Concurrent Write (CW) - no restriction on writing.
Usually, CW is unsafe.
Ways for solving contention among multiple writers:

- Arbitrary - no specific rule on which processor gets write-priority.
- Garbage - garbage (random values) becomes written at the location.
- Priority - the processors are given priorities (i.e., 1 to n) and the value written by the highest priority processor is the result.
- Combining - combine values being written (add, max, logical AND/OR).

PRAM model

Pros:
- abstract away from network, including dealing with varieties of protocols
- most ideas translate to other models

Cons:
- unrealistic memory model (not always constant time for access)
- in the synchronous version, lockstep removes much flexibility and restricts programming practices
- fixed number of processors

Work-depth models

Work-depth model are **more abstract** than multiprocessor models: there are no machine-dependent details to complicate the design and analysis of algorithms.

The cost of an algorithm is determined by examining the total number of operations that it performs, and the dependencies among those operations.

An important design goal is to have many tasks with as few dependencies as possible.

An algorithm's **work *W*** is the total number of operations that it performs; its **depth *D*** is the longest chain of dependencies among its operations.

We call the ratio ***P* = *W*/*D*** the **parallelism of the algorithm**.

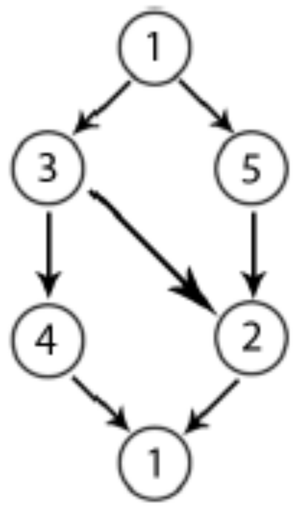
Direct acyclic graph (DAG) model

The DAG model (aka task graph model) is a special type of work-depth model.

A DAG is a graph with directed edges and no cycles.

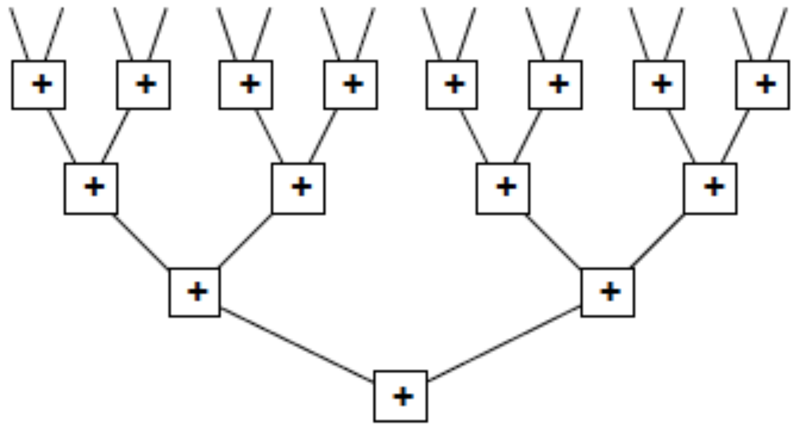
To model a parallel computation as a DAG, do the following.
• Represent small segments of sequential instructions as nodes of the graph. Use nonnegative node weights to show computation costs.
• Use edges to show dependencies between nodes. Optionally, use nonnegative edge weights to show communication costs.

Metrics:
• **Work *W* - sum of node weights**
• **Depth *D* - weight of the heaviest path in the DAG**



Work = 16
Depth = 9

Direct acyclic graph (DAG) model



Example

```
ALGORITHM: SUM($A$)
 1  if $|A| = 1$ then return $A[0]$
 2  else return SUM($\{A[2i] + A[2i+1] : i \in [0..|A|/2)\}$)
```

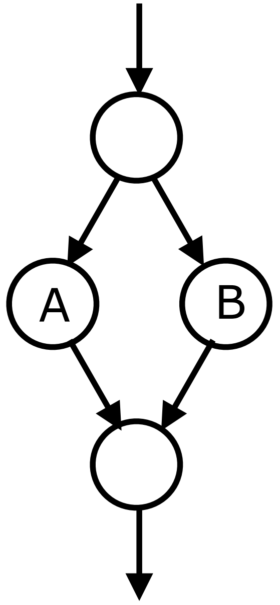
The *A[2i]+A[2i+1]* operations are performed in parallel, for each SUM() invocation.

W(n) = 2W(n/2)+1 where 1 is the constant work of the if
W(1) = O(1) constant time (may be larger than 1)
thus *W(n) = O(n)*

$D(n) = D(\lceil n/2 \rceil) + O(1) = O(\log n)$

Direct acyclic graph (DAG) model

Two operations A and B are **ordered** if there is a path in the DAG from A to B (A≺B) or from B to A (B≺A).



A **sequential ordering** of a parallel algorithm is any instruction-by-instruction execution of the parallel algorithm that does not violate the constraints imposed by the DAG.

Two operations A and B are **concurrent** iff they are not ordered.

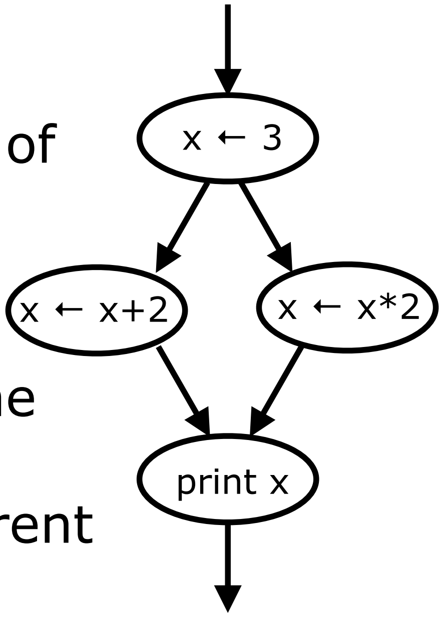
Concurrent Read (CR): two concurrent operations read from the same memory location.
Concurrent Write (CW): two concurrent operations write to the same memory location.
Concurrent Read-Write (CRW): two concurrent operations access the same location; one does a read, and the other a write.

Direct acyclic graph (DAG) model

Correctness of a parallel algorithm is important. If an algorithm doesn’t work, there is no point to optimize it.

Race conditions are conditions where the order of execution of instructions in the parallel machine affects the outcome of the computation.

If concurrent instructions happen to read and write to the same memory location, then the order in which they are executed dictates the values that have been read/written by the concurrent instructions.



A parallel algorithm is **race free** if it has no CWs or CRWs.

Performance

Performance

Speedup

$$S_n = T_s / T_n$$

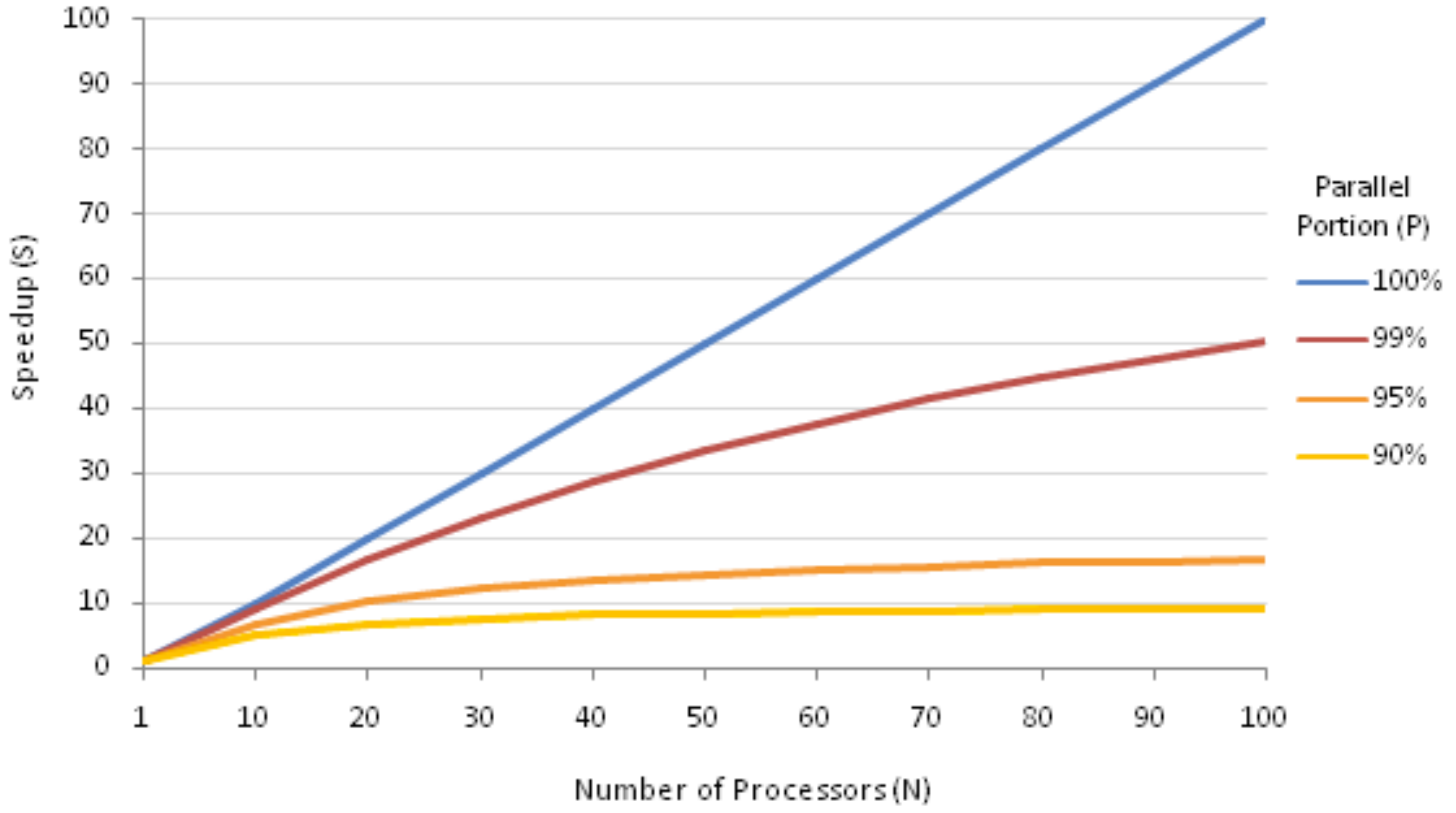
where
- n is the number of processors
- T_s is the execution time of the sequential version of the algorithm
- T_n is the execution time of the parallel algorithm, assuming n processors

Ideally: $S_n = n$

Amdahl's law: $S_n = n / (fn + (1-f))$
where f is the fraction of the algorithm that is strictly sequential

Performance

Amdahl's law



Performance

Efficiency

$$E_n = S_n / n$$

where

- n is the number of processors
- S_n is the speedup

Ideally: $E_n = 1$

Worst case: $E_n = 1/n$

With the work-depth formalism, a parallel algorithm is **work efficient** if W is asymptotically the same as the best known T_s for a sequential algorithm for the same problem.
When designing a parallel algorithm, make sure it is work efficient (or close to it) and minimize the depth. In other words, **maximize W/D**.

Performance

Parallel Overhead

It is the time needed to coordinate parallel tasks, as opposed to doing useful work.

The overhead may include the following factors:

- Task initialization time
- Synchronization time
- Data communications
- Software overhead imposed by compilers, libraries, management tools, operating system
- Task completion time

Performance

Granularity

Granularity is a qualitative measure of the relation between processing and communication.

Coarse Grain: a significant amount of processing is done between two communication events.

Fine Grain: a limited amount of processing is executed between two communication events.

Performance

Scalability

It is the ability of a parallel system (hardware and/or software) to offer a performance improvement that is proportional to the number of processors.

Factors that affect scalability are:

- the bandwidth of the bus between memory and CPU
- the bandwidth and the delay of the network
- the parallel algorithm and its implementation
- the parallel overhead

Considering the work-depth formalism, a parallel algorithm is **scalably parallel** if $W/D \to \infty$ when the input size $\to \infty$.

Design patterns

Guidelines for parallel algorithm design

1) Postpone consideration of the details of the computational platform until after the decomposition phase.

A design that is tied to a particular architecture is unlikely to perform well on other machines, whereas an abstract design based on tasks and dependencies can be adapted to run on any machine.

2) Design many independent tasks, whose number increases with the problem size.

Resist the temptation to think of tasks as cores!

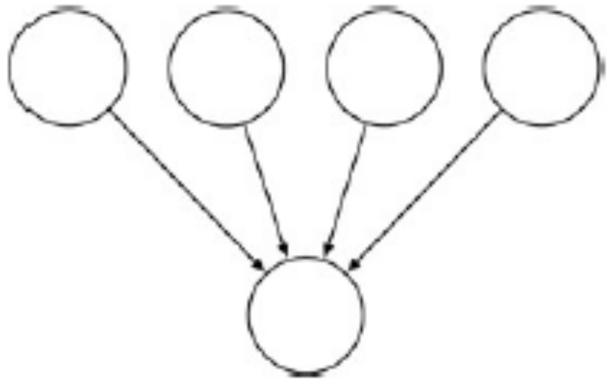
Patterns for parallel algorithm design

The fundamental activity of parallel algorithm design is **decomposition**, which means identifying tasks and their dependencies.

In parallel algorithm design, there are many general patterns that can be used across a variety of problem areas. Major patterns are listed below. In the following, we will discuss some of them.

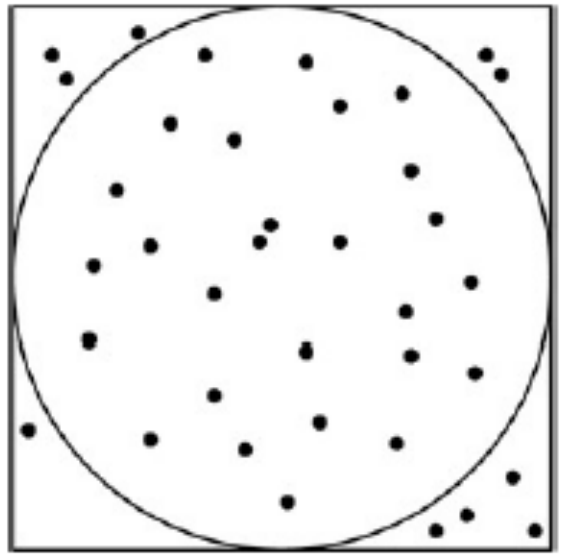
- **Embarrassingly parallel**
- Reduction
- Scan
- **Divide-and-conquer**
- Pipeline
- **Data decomposition**
- **Randomization**

Embarrassingly parallel



This pattern applies to cases where the **decomposition into independent tasks is self-evident**. Each task does its own computation independent of the other tasks, and the resulting task graph is completely disconnected.

In Monte Carlo methods, for example, a problem is solved by conducting numerous experiments with pseudo-randomly selected variables.



The classic illustrative example is the **approximation of π**. A circle of radius 1 is inscribed in a square and points are randomly chosen inside the square. The area of the circle, and hence π, can be estimated as the fraction of points that lie in the circle multiplied by the area of the square. This problem can be decomposed into independent tasks that each generate a point and determine whether it lies in the circle.

Divide-and-conquer

A problem is solved by recursively **dividing** it into two or more subproblems of the same type, until they become simple enough to be **solved** directly.

The solutions to the subproblems are then **merged** to construct a solution to the original problem.

Because the subproblems created are typically independent, they can be solved in parallel. The task decomposition can be decided dynamically at run-time, in order to keep the best parallelism degree (especially when it is difficult to predict the cost of each task).

In order for divide-and-conquer to yield a highly parallel algorithm, it is often necessary to parallelize the divide step and the merge step.

Divide-and-conquer

Example - **Mergesort**

Input: a sequence of *n* values
Output: the *n* values in sorted order

Main idea:
- split the sequence into two sequences of $n/2$ values
- recursively sort each sequence: $2T_s(n/2)$
- merge the two sorted sequences: $O(n)$

Sequential running time:

$$T_s(n) = \begin{cases} 2T_s(n/2) + O(n) & n>1 \\ O(1) & n=1 \end{cases}$$

--> $T_s(n) = O(n\log n)$

Divide-and-conquer

Example - **Mergesort**

Basic parallel approach:
- recursively sort each sequence in parallel, sequential merging

$W(n) = 2W(n/2) + O(n) = O(n\log n)$ the same as $T_s(n)$
$D(n) = D(n/2) + O(n) = O(n)$

--> parallelism $W/D = O(\log n)$ not very much...

Advanced parallel approach:
- use pipelined merge [Cole 1986]

$W(n) = O(n\log n)$
$D(n) = O(\log n)$

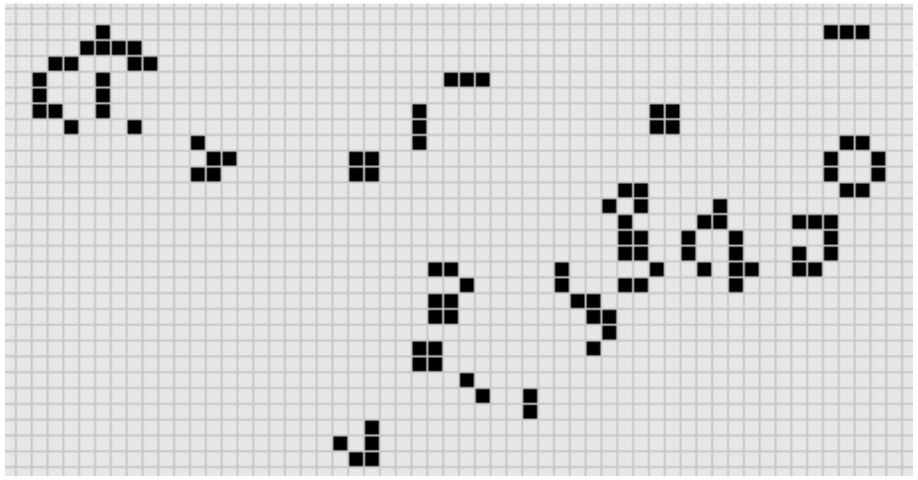
--> parallelism $W/D = O(n)$ much better!

Data decomposition

Rather than focusing on task decomposition, sometimes it is easier and more effective to **decompose the data structures**. Then, a different instance of the same task is associated with each portion. **Each task does the same work, but on distinct data.**

Examples:

- grid applications like cellular automata (e.g., Conway's Game of Life)
- matrix-vector multiplication
- image processing



Randomization

Random numbers are used in parallel algorithms to ensure that processors can make local decisions which, with high probability, add up to good global decisions. Here we consider three uses of randomness.

Sampling: use randomness to select a representative sample from a set of input elements, for partitioning or pruning purposes [Raman1998].

Symmetry breaking: use randomness to select a subset of independent operations from a large set of symmetrical operations, where interdependencies prevent simultaneous execution of all the operations in the set.

Example: finding a maximal independent set of a graph [Luby1986]

Load Balancing: use randomness to evenly assign tasks to processors.

References

• E. Aubanel, *Elements of Parallel Computing*, CRC Press, 2017

• T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to algorithms*, MIT Press, 3rd edition, 2009.

• G. E. Blelloch, B. M. Maggs, *Parallel Algorithms*, The Computer Science Handbook, 2004

• R. Cole, *Parallel Merge Sort*, SIAM Journal on Computing, vol. 17, no. 4, 1986

• R. Raman, *Random Sampling Techniques in Parallel Computation*, International Parallel Processing Symposium (IPPS), Orlando, Florida, March 1998

• M. Luby, *A Simple Parallel Algorithm for the Maximal Independent Set Problem*, 17th Annual ACM Symposium on Theory of Computing (STOC '85), Providence, Rhode Island, May 1985